Software Requirements and Design Specification

1. System Services CSCI

1.1 CORBA Services Introduction

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them.

The Object Request Broker (ORB) is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.[1]

1.1.1 CORBA Services Overview

Applications can use the CORBA services transparently. This means that any function or method within a program maybe called as normal. The implementation for that function may or may not reside within the same process, machine, or network; however, this is hidden from the client program. Calls can be blocking or non-blocking. Arguments used within the function calls are marshaled and unmarshaled transparently. The arguments are also independent of machine size and byte ordering since all CORBA interfaces are specified with CORBA types, which map to normal C++ types. Refer to figure 1.

CORBA transparency

All CORBA objects within the system may register with the naming service. Applications can then bind to a specific object using its name. This mechanism allows clients to connect to servers without any knowledge of their location . CORBA objects not within the directory can still be accessed if the caller uses an object reference. Details of implementing the call are handled automatically.

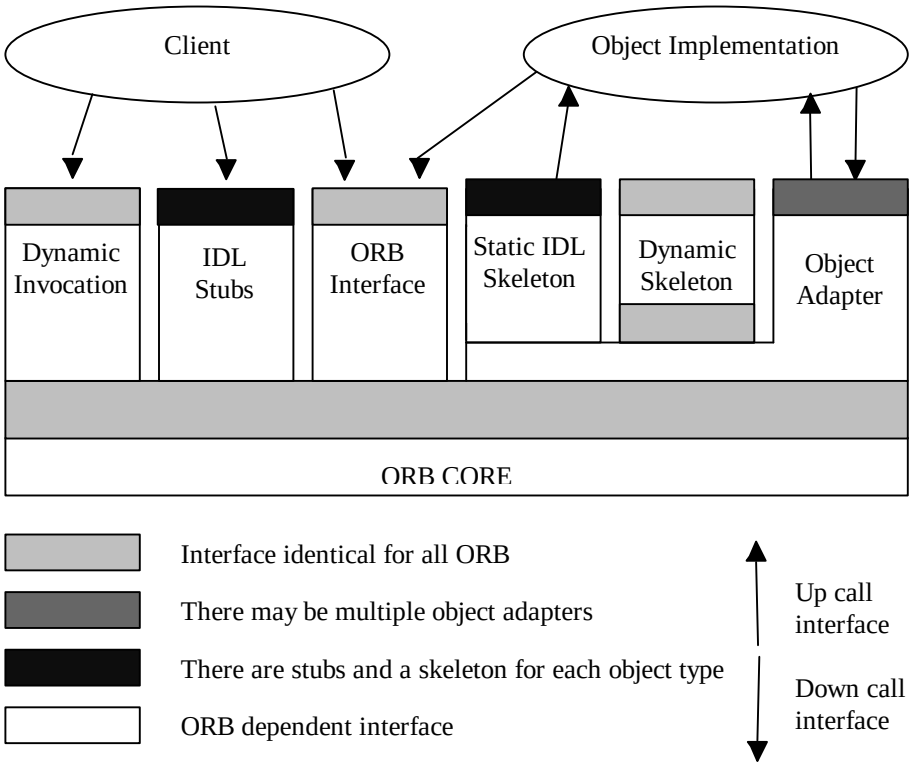
Note: The term "CORBA Services" used in the headings of this document does not refer to the OMG concept of "CORBA Services", which is used to describe functionality outside those features described in the core specification.

1.1.2 CORBA Operational Description

The diagram below shows the basic structure of the CORBA specification. The relationship of the client and server objects with respect to the ORB and the functionality implemented to provide the communication infrastructure are similar across all CORBA compliant products.

[1] http://www.omg.org/about/wicorba.htm (10/30/97)

Software Requirements and Design Specification



ORB Architecture

- Object Implementation -- This defines operations that implement a CORBA IDL interface. Object implementations can be written in a variety of languages including C, C++, Java, Smalltalk, and Ada.

- Client -- This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, i.e., obj->op(args). The remaining components in the above figure help to support this level of transparency.

- Object Request Broker (ORB) -- The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.

- ORB Interface -- An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

- CORBA IDL stubs and skeletons -- CORBA IDL stubs and skeletons serve as the ``glue" between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler. The use

of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

- Dynamic Invocation Interface (DII) -- This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking deferred synchronous (separate send and receive operations) and oneway (send-only) calls.

- Dynamic Skeleton Interface (DSI) -- This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

- Object Adapter -- This assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects). [2]

1.2 CORBA Services Specifications

1.2.1 CORBA Services Groundrules

CORBA has the following assumptions and constraints.

- Interfaces are defined in IDL (Interface Definition Language).
- Native code is generated by an IDL compiler.
- The application must register its objects with the CORBA ORB.
- All communication is done through the ORB.

1.2.2 CORBA Services Functional Requirements

1. CORBA services will provide the ability to send and receive object messages.
2. CORBA services will allow users to implement exception handling for object messaging.
3. CORBA services will provide a method to manage the creation and destruction of objects within the ORB.
4. The CORBA implementation will provide multi-threaded libraries for the client and server applications.
5. The CORBA implementation will provide the ability to log CORBA messages to the SDC.
6. The CORBA implementation will provide the ability to interface with command authentication.
7. *Provide a mechanism to migrate services to redundant servers. (post THOR)*

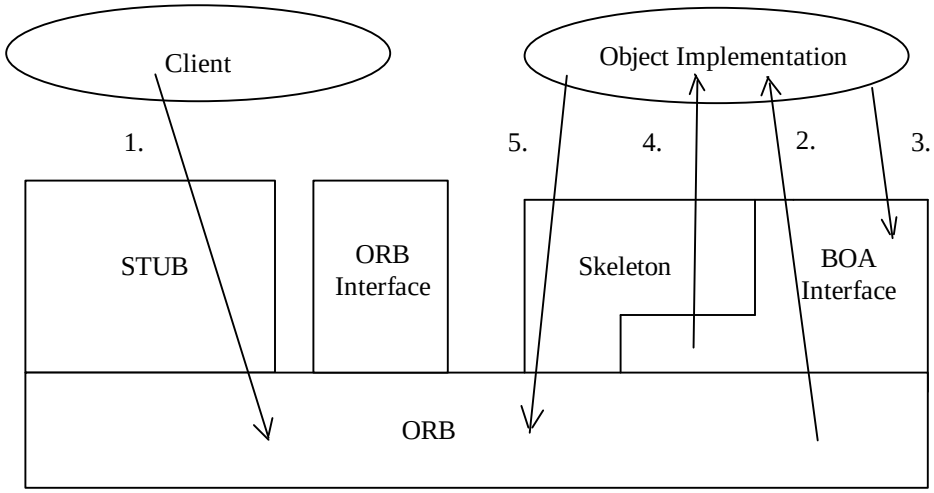
1.2.3 CORBA Performance Requirements

None

[2] http://www.cs.wustl.edu/~schmidt/corba-overview.html (10/29/97)

1.2.4 Detailed Data Flow

The client invokes methods within the server through the following call path. The data within the call follows this sequence of events. Below is a diagram showing the overall flow.



Call Path through ORB

Invocation follows the following call-path through the ORB:

1. Client calls method through stub.
2. ORB hands request to the BOA, which activates the implementation.
3. Implementation invokes the BOA to say it is active and available.
4. BOA passes the method request into Implementation via skeleton.
5. Implementation returns result (or exception) back through the ORB.[3]

1.2.5 CORBA Services Design Specification

CORBA Product implementation

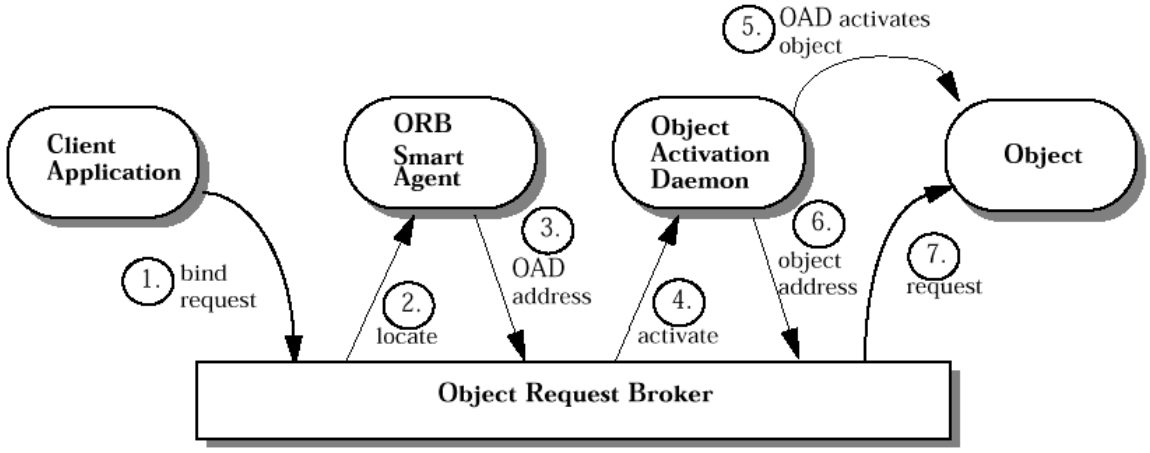
The CORBA services will be provided by Visigenic's Visibroker V3.0 for C++. This product is compliant with the OMG 2.1 CORBA standard and has the following characteristics.

- Visigenic uses a smart agent (osagent) to provide a means for servers and clients to register and find objects. A client or server uses a broadcast packet to locate an agent, then uses point to point communication thereafter. The smart agent will reside on the Master CCP.
- For specific blocking calls there are parameters that can be tuned to specify time-out options, such as: send, receive, and connection.
- The BOA is responsible for handling communication between the object implementation (server) and the ORB. Visigenic implements some of the BOA (Basic Object Adapter) features using an OAD process. This process can start a secondary process, if needed, to provide the objects required to honor

[3] http://www.qds.com/people/apope/ap_TheObject.html (10/29/97)

a client requests. Note: Future "Object Adapters" maybe available for other types of objects, for example those inside a database.

- For C++ there are different types of inter process communication used depending on the relation between the client and server objects. If they both reside in the same process then a virtual call is made. If the client and server are on the same machine, but in different processes, then shared memory is used. If the client and server are on separate machines then the TCP/IP networking protocol will be employed.



Overview of an object invocation

The basic steps to creating a client and server are:

A. Define the interface between the client and server.
B. Use the IDL compiler to generate client and server stubs.
C. Fill in the server side skeleton.
D. Run the osagent somewhere within the LAN that will contain your client and server, multiple network communications require extra configuration.

Steps to implement an object (server):

A. Initialize the ORB object.
B. Initialize the BOA object.
C. Create the object to be used by the clients.
D. Register the object with the BOA.
E. Wait for calls.

Steps required to "use" an object within the ORB:

A. Initialize the ORB object.
B. Bind to a server object within the ORB.
C. Invoke the method on that object.

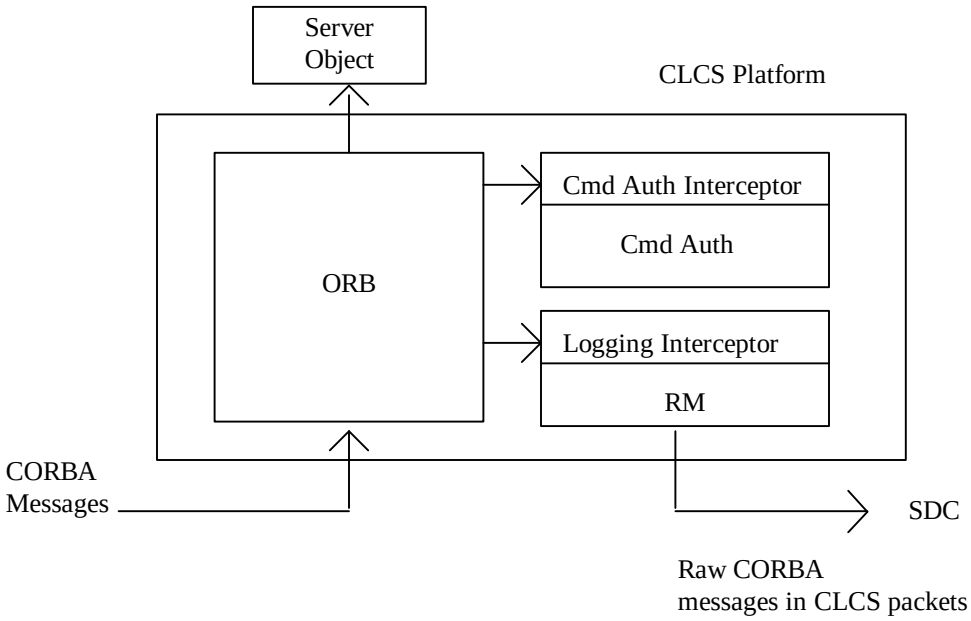
SDC Logging

In order to provide the logging capability for the SDC additional software will be written. The Visigenic product has implemented the concept of "interceptors". A message interceptor provides the ability to

access the CORBA message before it is handed to the server object. Therefore extra code will be used at the "interceptor" point to bundle the object message and multicast it into the SDC via RM.

Command Authentication

The interceptor capability will also be used to implement command authorization into the CORBA methods. The Command Authentication Interceptor will invoke a command API with the information required for authentication. Below is a figure showing the interceptors for logging and command authentication.



CLCS additions to the product

1.2.6 CORBA Name External Interfaces

N/A

1.2.7 Recorded Data

Name of Recorded Data	*Recording Type*	*SDC*	*Local*
CORBA Messages		X	

1.2.8 CSC Name Printer Formats

N/A

1.2.9 CSC Name Table Formats

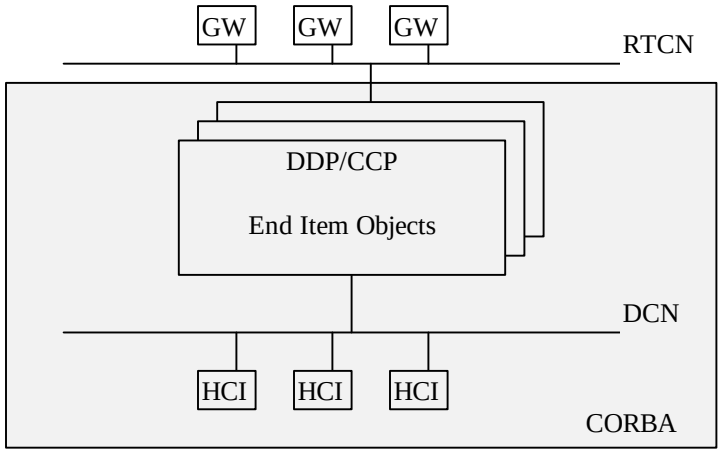
N/A

1.2.10 CORBA Services Test Plan

Any test plan involving this product would need to reflect the architectural implementation used. The assumption will be that the majority of the objects within the ORB are using the centralized CCP and DDP resources. For example if the CCP and DDP were to contain "factories" that produced "end item" objects, the ORB could contain several thousand objects. In fact there could be entire sets of hierarchical objects in place to command, examine, and otherwise control particular "end item" objects. Given this type of architecture it would be advisable to test basic system resources, such as memory and processor cycles being consumed. It would also be advisable to relate this type of implementation to overall network traffic. A possible approach might be to define a "use case" mode to estimate the number of events and messages which maybe generated by the above implementation.

- Test memory utilization of system with large number of objects within ORB.
- Test CCP and DDP for ability to generate and maintain objects.
- Test latency between HCI and DDP/CCP machines.
- Test security of objects with calls made by "unauthorized" clients.
- Test ability of ORB to process large number of messages concurrently.
- Test for possible deadlock or race conditions within distributed system.
- Test ability to log CORBA messages.

1.2.11 CORBA System Diagram



CORBA within the DCN

The above diagram shows that all the machines within the DCN network can implement or call objects through the ORB. It maybe that most of the objects within the ORB will be implemented on the DDP/CCP machines and most of the client invocations will originate from the HCI workstations.